Apache HTTP Server Version 1.3

Is this the version you want? For more recent versions, check our documentation index.

Apache 1.3 Dynamic Shared Object (DSO) Support

Originally written by
Ralf S. Engelschall <rse@apache.org>, April 1998

Background

On modern Unix derivatives there exists a nifty mechanism usually called dynamic linking/loading of *Dynamic Shared Objects* (DSO) which provides a way to build a piece of program code in a special format for loading it at run-time into the address space of an executable program.

This loading can usually be done in two ways: Automatically by a system program called `ld.so` when an executable program is started or manually from within the executing program via a programmatic system interface to the Unix loader through the system calls `dlopen()/dlsym()`.

In the first way the DSO's are usually called *shared libraries* or *DSO libraries* and named `libfoo.so` or `libfoo.so.1.2`. They reside in a system directory (usually `/usr/lib`) and the link to the executable program is established at build-time by specifying `-lfoo` to the linker command. This hard-codes library references into the executable program file so that at start-time the Unix loader is able to locate `libfoo.so` in `/usr/lib`, in paths hard-coded via linker-options like `-R` or in paths configured via the environment variable `LD_LIBRARY_PATH`. It then resolves any (yet unresolved) symbols in the executable program which are available in the DSO.

Symbols in the executable program are usually not referenced by the DSO (because it's a reusable library of general code) and hence no further resolving has to be done. The executable program has no need to do anything on its own to use the symbols from the DSO because the complete resolving is done by the Unix loader. (In fact, the code to invoke `ld.so` is part of the run-time startup code which is linked into every executable program which has been bound non-static). The advantage of dynamic loading of common library code is obvious: the library code needs to be stored only once, in a system library like `libc.so`, saving disk space for every program.

In the second way the DSO's are usually called *shared objects* or *DSO files* and can be named with an arbitrary extension (although the canonical name is `foo.so`). These files usually stay inside a program-specific directory and there is no automatically established link to the executable program where they are used. Instead the executable program manually loads the DSO at run-time into its address space via `dlopen()`. At this time no resolving of symbols from the DSO for the executable program is done. But instead the Unix loader automatically resolves any (yet unresolved) symbols in the DSO from the set of symbols exported by the executable program and its already loaded DSO libraries (especially all symbols from the ubiquitous `libc.so`). This way the DSO gets knowledge of the executable program's symbol set

as if it had been statically linked with it in the first place.

Finally, to take advantage of the DSO's API the executable program has to resolve particular symbols from the DSO via `dlsym()` for later use inside dispatch tables *etc.* In other words: The executable program has to manually resolve every symbol it needs to be able to use it. The advantage of such a mechanism is that optional program parts need not be loaded (and thus do not spend memory) until they are needed by the program in question. When required, these program parts can be loaded dynamically to extend the base program's functionality.

Although this DSO mechanism sounds straightforward there is at least one difficult step here: The resolving of symbols from the executable program for the DSO when using a DSO to extend a program (the second way). Why? Because "reverse resolving" DSO symbols from the executable program's symbol set is against the library design (where the library has no knowledge about the programs it is used by) and is neither available under all platforms nor standardized. In practice the executable program's global symbols are often not re-exported and thus not available for use in a DSO. Finding a way to force the linker to export all global symbols is the main problem one has to solve when using DSO for extending a program at run-time.

Windows and NetWare provide similar facilities, although they are implemented somewhat differently than the description of Unix DSO throughout this document. In particular, DSO modules (DLL's and NLM's, respectively) are built quite differently than their Unix cousins. This document does not attempt to explore the topic of building DSO modules on these platforms. The description of mod_so and its configuration, however, are similar.

Practical Usage

The shared library approach is the typical one, because it is what the DSO mechanism was designed for, hence it is used for nearly all types of libraries the operating system provides. On the other hand using shared objects for extending a program is not used by a lot of programs.

As of 1998 there are only a few software packages available which use the DSO mechanism to actually extend their functionality at run-time: Perl 5 (via its XS mechanism and the DynaLoader module), Netscape Server, *etc.* Starting with version 1.3, Apache joined the crew, because Apache already uses a module concept to extend its functionality and internally uses a dispatch-list-based approach to link external modules into the Apache core functionality. So, Apache is really predestined for using DSO to load its modules at run-time.

As of Apache 1.3, the configuration system supports two optional features for taking advantage of the modular DSO approach: compilation of the Apache core program into a DSO library for shared usage and compilation of the Apache modules into DSO files for explicit loading at run-time.

Implementation

The DSO support for loading individual Apache modules is based on a module named `mod_so.c` which has to be statically compiled into the Apache core. It is the only module besides `http_core.c` which cannot be put into a DSO itself (bootstrapping!). Practically all other distributed Apache modules can then be placed into a DSO by individually enabling the DSO build for them via `configure`'s `--enable-shared` option (see top-level `INSTALL` file) or by changing the `AddModule` command in your `src/Configuration` into a `SharedModule` command (see `src/INSTALL` file). After a module is compiled into a DSO named `mod_foo.so` you can use `mod_so`'s `LoadModule` command in your `httpd.conf` file to load this module at server startup or restart.

To simplify this creation of DSO files for Apache modules (especially for third-party modules) a new

support program named apxs (*APache eXtenSion*) is available. It can be used to build DSO based modules *outside of* the Apache source tree. The idea is simple: When installing Apache the `configure`'s `make install` procedure installs the Apache C header files and puts the platform-dependent compiler and linker flags for building DSO files into the `apxs` program. This way the user can use `apxs` to compile his Apache module sources without the Apache distribution source tree and without having to fiddle with the platform-dependent compiler and linker flags for DSO support.

To place the complete Apache core program into a DSO library (only required on some of the supported platforms to force the linker to export the apache core symbols -- a prerequisite for the DSO modularization) the rule `SHARED_CORE` has to be enabled via `configure`'s `--enable-rule=SHARED_CORE` option (see top-level `INSTALL` file) or by changing the `Rule` command in your `Configuration` file to `Rule SHARED_CORE=yes` (see `src/INSTALL` file). The Apache core code is then placed into a DSO library named `libhttpd.so`. Because one cannot link a DSO against static libraries on all platforms, an additional executable program named `libhttpd.ep` is created which both binds this static code and provides a stub for the `main()` function. Finally the `httpd` executable program itself is replaced by a bootstrapping code which automatically makes sure the Unix loader is able to load and start `libhttpd.ep` by providing the `LD_LIBRARY_PATH` to `libhttpd.so`.

Supported Platforms

Apache's `src/Configure` script currently has only limited but adequate built-in knowledge on how to compile DSO files, because as already mentioned this is heavily platform-dependent. Nevertheless all major Unix platforms are supported. The definitive current state (May 1999) is this:

- Out-of-the-box supported platforms:
  (actually tested versions in parenthesis)

```
o  FreeBSD              (2.1.5, 2.2.x, 3.x, 4.x)
o  OpenBSD              (2.x)
o  NetBSD               (1.3.1)
o  BSDI                 (3.x, 4.x)
o  Linux                (Debian/1.3.1, RedHat/4.2)
o  Solaris              (2.4, 2.5, 2.6, 2.7)
o  SunOS                (4.1.3)
o  Digital UNIX         (4.0)
o  IRIX                 (5.3, 6.2)
o  HP/UX                (10.20)
o  UnixWare             (2.01, 2.1.2)
o  SCO                  (5.0.4)
o  AIX                  (3.2, 4.1.5, 4.2, 4.3)
o  ReliantUNIX/SINIX    (5.43)
o  SVR4                 (-)
o  Mac OS X Server      (1.0)
o  Mac OS               (10.0 preview 1)
o  OpenStep/Mach        (4.2)
o  DGUX                 (??)
o  NetWare              (5.1)
o  Windows              (95, 98, NT 4.0, 2000)
```

- Explicitly unsupported platforms:

```
o  Ultrix               (no dlopen-style interface under this platform)
```

Usage Summary

To give you an overview of the DSO features of Apache 1.3, here is a short and concise summary:

1. Placing the Apache core code (all the stuff which usually forms the `httpd` binary) into a DSO `libhttpd.so`, an executable program `libhttpd.ep` and a bootstrapping executable program `httpd` (Notice: this is only required on some of the supported platforms to force the linker to export the Apache core symbols, which in turn is a prerequisite for the DSO modularization):
   - Build and install via `configure` (preferred):

     ```
     $ ./configure --prefix=/path/to/install
                   --enable-rule=SHARED_CORE ...
     $ make install
     ```

   - Build and install manually:

     ```
     - Edit src/Configuration:
       << Rule SHARED_CORE=default
       >> Rule SHARED_CORE=yes
       << EXTRA_CFLAGS=
       >> EXTRA_CFLAGS= -DSHARED_CORE_DIR=\"/path/to/install/libexec\"
     $ make
     $ cp src/libhttpd.so* /path/to/install/libexec/
     $ cp src/libhttpd.ep  /path/to/install/libexec/
     $ cp src/httpd        /path/to/install/bin/
     ```

2. Build and install a *distributed* Apache module, say `mod_foo.c`, into its own DSO `mod_foo.so`:
   - Build and install via `configure` (preferred):

     ```
     $ ./configure --prefix=/path/to/install
             --enable-shared=foo
     $ make install
     ```

   - Build and install manually:

     ```
     - Edit src/Configuration:
       << AddModule    modules/xxxx/mod_foo.o
       >> SharedModule modules/xxxx/mod_foo.so
     $ make
     $ cp src/xxxx/mod_foo.so /path/to/install/libexec
     - Edit /path/to/install/etc/httpd.conf
       >> LoadModule foo_module /path/to/install/libexec/mod_foo.so
     ```

3. Build and install a *third-party* Apache module, say `mod_foo.c`, into its own DSO `mod_foo.so`
   - Build and install via `configure` (preferred):

     ```
     $ ./configure --add-module=/path/to/3rdparty/mod_foo.c
             --enable-shared=foo
     $ make install
     ```

   - Build and install manually:

     ```
     $ cp /path/to/3rdparty/mod_foo.c /path/to/apache-1.3/src/modules/extra/
     - Edit src/Configuration:
       >> SharedModule modules/extra/mod_foo.so
     $ make
     $ cp src/xxxx/mod_foo.so /path/to/install/libexec
     - Edit /path/to/install/etc/httpd.conf
       >> LoadModule foo_module /path/to/install/libexec/mod_foo.so
     ```

4. Build and install a *third-party* Apache module, say `mod_foo.c`, into its own DSO `mod_foo.so` *outside of* the Apache source tree:
   - Build and install via apxs:

     ```
     $ cd /path/to/3rdparty
     $ apxs -c mod_foo.c
     $ apxs -i -a -n foo mod_foo.so
     ```

Advantages & Disadvantages

The above DSO based features of Apache 1.3 have the following advantages:

- The server package is more flexible at run-time because the actual server process can be assembled at run-time via `LoadModule` `httpd.conf` configuration commands instead of `Configuration` `AddModule` commands at build-time. For instance this way one is able to run different server instances (standard & SSL version, minimalistic & powered up version [mod_perl, PHP3], *etc.*) with only one Apache installation.
- The server package can be easily extended with third-party modules even after installation. This is at least a great benefit for vendor package maintainers who can create a Apache core package and additional packages containing extensions like PHP3, mod_perl, mod_fastcgi, *etc.*
- Easier Apache module prototyping because with the DSO/`apxs` pair you can both work outside the Apache source tree and only need an `apxs -i` command followed by an `apachectl restart` to bring a new version of your currently developed module into the running Apache server.

DSO has the following disadvantages:

- The DSO mechanism cannot be used on every platform because not all operating systems support dynamic loading of code into the address space of a program.
- The server is approximately 20% slower at startup time because of the symbol resolving overhead the Unix loader now has to do.
- The server is approximately 5% slower at execution time under some platforms because position independent code (PIC) sometimes needs complicated assembler tricks for relative addressing which are not necessarily as fast as absolute addressing.
- Because DSO modules cannot be linked against other DSO-based libraries (`ld -lfoo`) on all platforms (for instance a.out-based platforms usually don't provide this functionality while ELF-based platforms do) you cannot use the DSO mechanism for all types of modules. Or in other words, modules compiled as DSO files are restricted to only use symbols from the Apache core, from the C library (`libc`) and all other dynamic or static libraries used by the Apache core, or from static library archives (`libfoo.a`) containing position independent code. The only chances to use other code is to either make sure the Apache core itself already contains a reference to it, loading the code yourself via `dlopen()` or enabling the `SHARED_CHAIN` rule while building Apache when your platform supports linking DSO files against DSO libraries.
- Under some platforms (many SVR4 systems) there is no way to force the linker to export all global symbols for use in DSO's when linking the Apache httpd executable program. But without the visibility of the Apache core symbols no standard Apache module could be used as a DSO. The only chance here is to use the `SHARED_CORE` feature because this way the global symbols are forced to be exported. As a consequence the Apache `src/Configure` script automatically enforces `SHARED_CORE` on these platforms when DSO features are used in the `Configuration` file or on the configure command line.

Apache HTTP Server